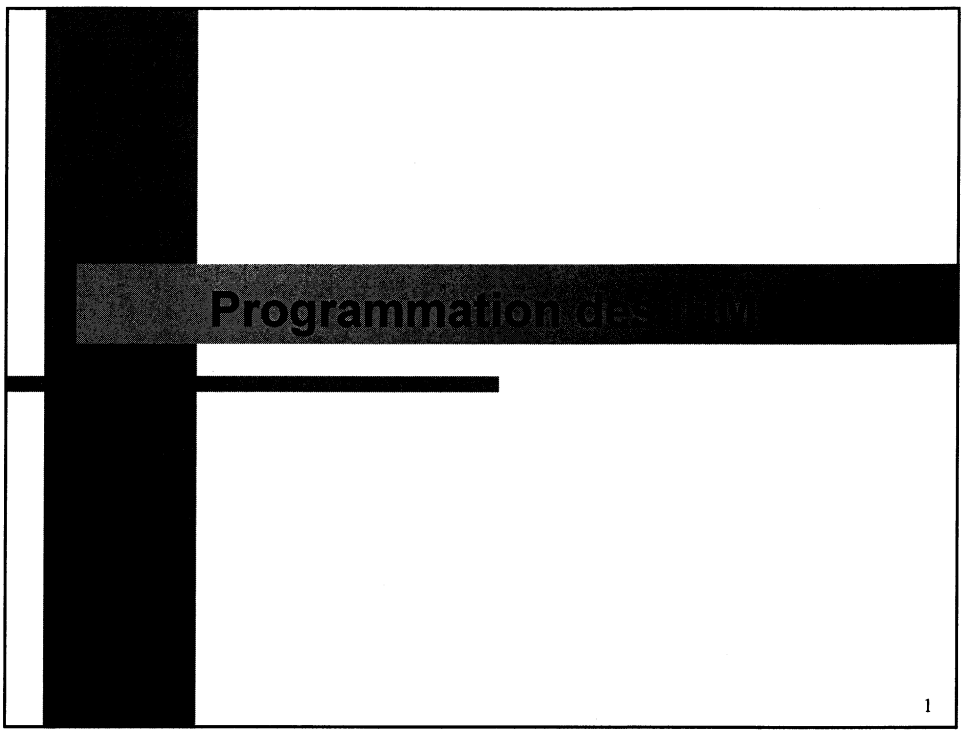
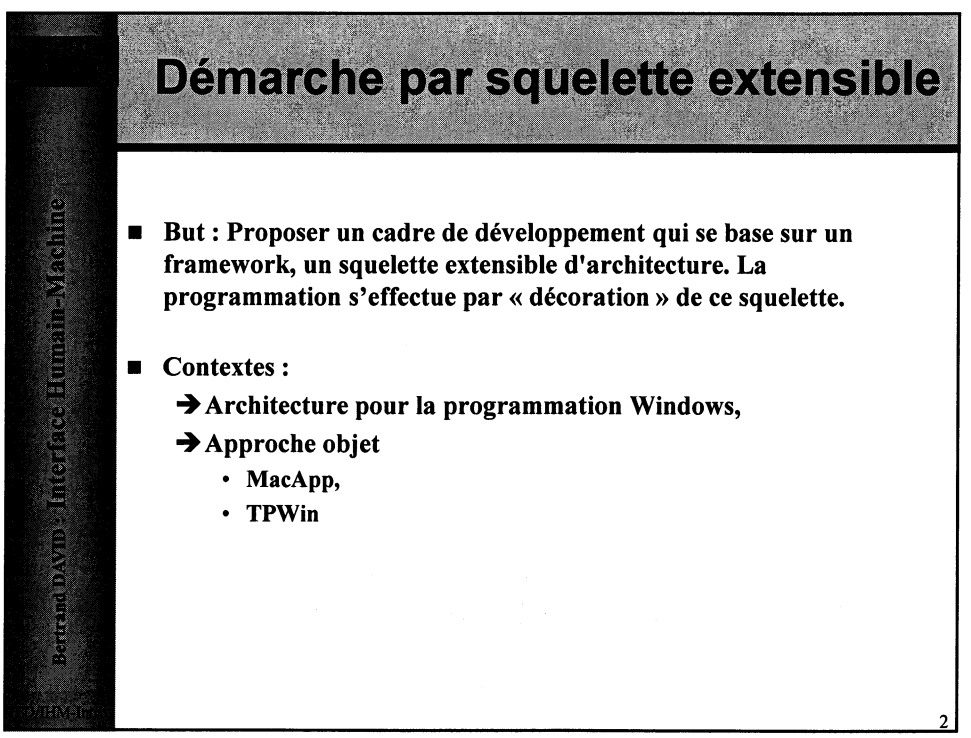

Programmation des IHM
1


Démarche par squelette extensible
Bertrand DAVID - Interface Humain-Machine
But : Proposer un cadre de développement qui se base sur un framework, un squelette extensible d'architecture. La programmation s'effectue par « décoration » de ce squelette.
Contextes :
Architecture pour la programmation Windows,
Approche objet
MacApp,
TPWin
2

Pourquoi WINDOWS ?

Différentes solutions:

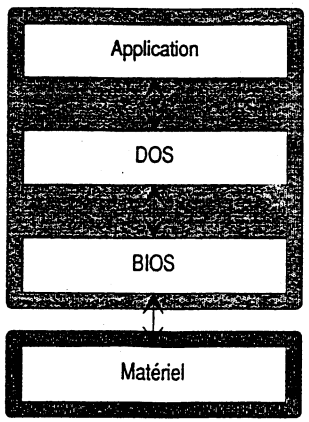
- **Approche classique: Bios-Dos-Appli**

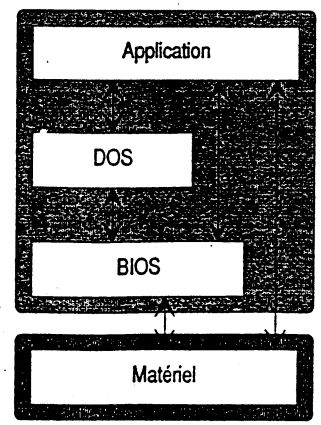
  avec comme problème: faible niveau de gestion du clavier et de l'écran

  ainsi qu'une impossibilité de modifier facilement le BIOS pour intégrer de nouveaux matériels

- **Approche spécifique: Contournement du DOS et/ou du BIOS**

  avec comme problème: la dépendance d'une configuration spécifique





Approche WINDOWS

Objectif: virtualisation des périphériques

Deux ensembles fonctionnels constituent l'architecture Windows:

- le noyau (indépendant du matériel)
- la machine virtuelle (réalisant le lien avec des dispositifs physiques)

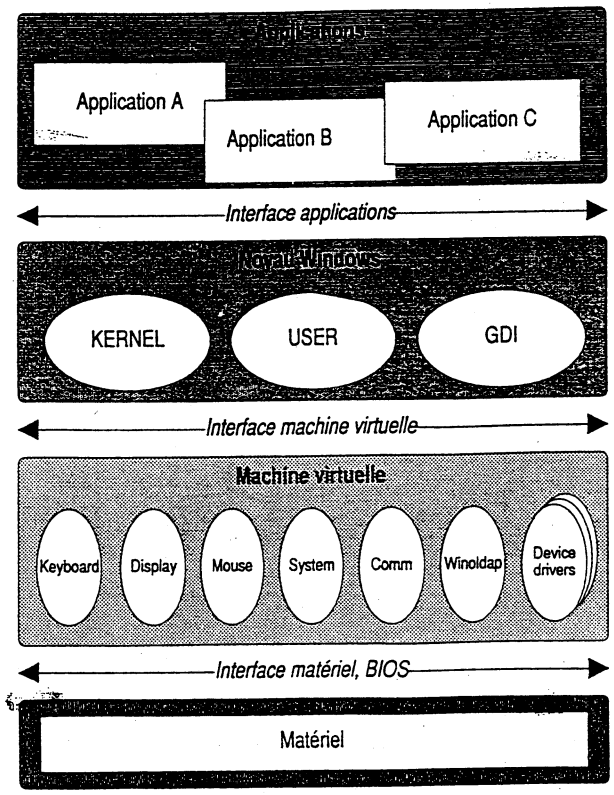
Le noyau comprend trois modules:

KERNEL, qui se charge du contrôle et de l'allocation des ressources:

- allocation mémoire
- chargement des applications
- gestion de multitâche

USER, qui assure la fonction de gestionnaire de fenêtre et de messages.

GDI (Graphics Device Interface), qui contient la bibliothèque graphique et permet de créer des images.



La machine virtuelle est constituée de plusieurs modules:

KEYBOARD est le module de gestion du clavier et du haut-parleur,

MOUSE est le module chargé de détecter les événements provoqués par la souris,

DISPLAY est le module qui traite les sorties de données vers l'écran de visualisation,

SYSTEM interface certaines parties matérielles du système telles que l'horloge ou les disques,

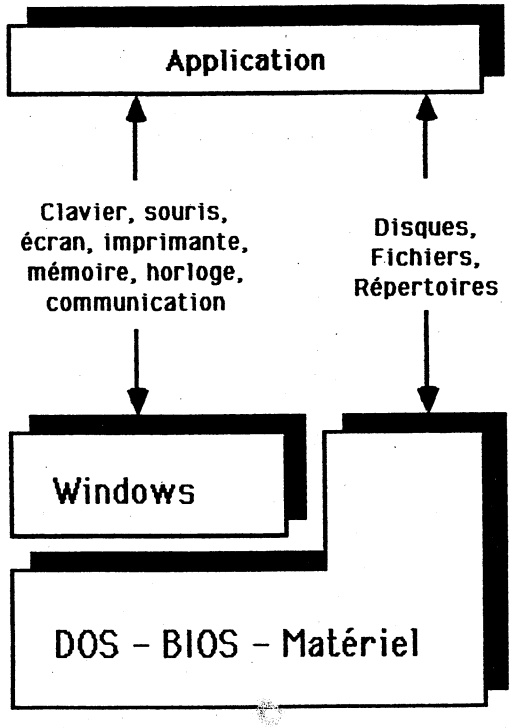
COMM contrôle les communications à travers les ports série (COM1, COM2,..) ou parallèle (LPT1, LPT2,...),

SOUND génère les sons sur le haut-parleur du système,

WINOLDAP est le module particulier qui a pour rôle de préparer l'environnement nécessaire pour le support des applications non Windows et d'en permettre l'exécution à partir de Windows.

LA PLACE DE WINDOWS

Gestion des périphériques mais pas de disques, de fichiers ou de répertoires



Qu'apporte WINDOWS ?

- le multi-tâche
- l'indépendance vis-à-vis du matériel
- la communication entre applications
- l'édition de liens dynamiques

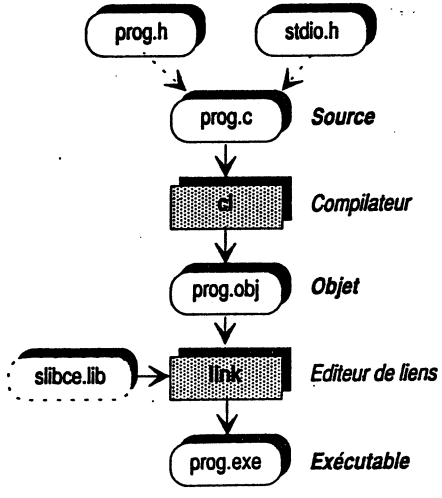
MS - WINDOWS II
Architecture d'une application

1. De DOS à WINDOWS

Les fichiers constituant une application:

- Application DOS:
  - un programme source .C
  - éventuellement un fichier .H
  - compilateur CL génère .OBJ
  - éditeur de liens LINK construit .EXE
  - on peut compliquer en ajoutant d'autres sources ou bibliothèques

Processus d'élaboration:



- Application sous Windows:

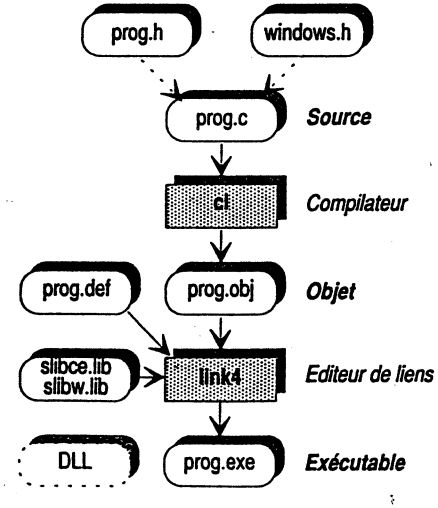
Première approche:

- un programme source .C
- des fichiers .H
- compilateur CL génère .OBJ
- éditeur de liens LINK4 construit .EXE en utilisant le fichier .DEF et des bibliothèques .LIB

Remarques:

- LINK4 gère l'édition de liens dynamique et fournit un en-tête d'un nouveau format pour le .EXE
- Le fichier .DEF contient des instructions liée à l'édition de liens dynamique et à la gestion de la mémoire

Processus d'élaboration:



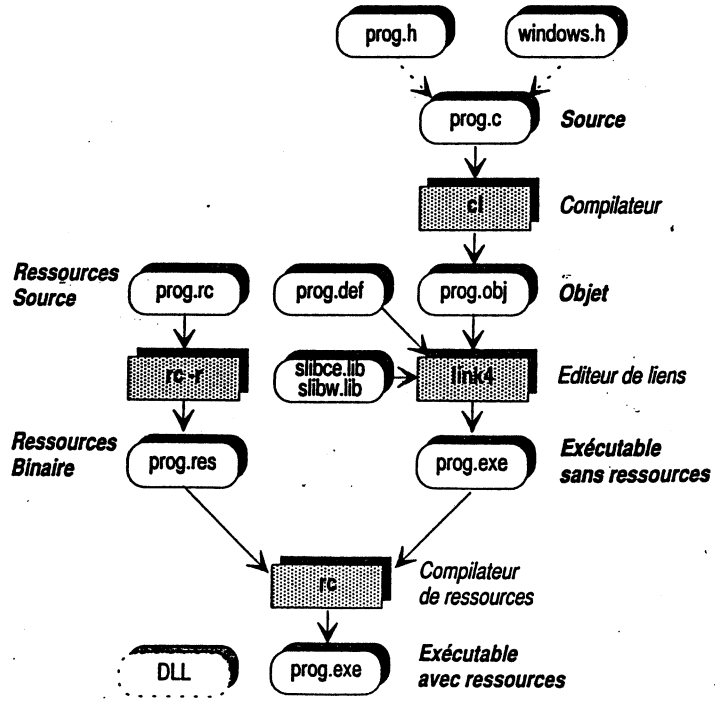
Deuxième approche: avec fichier des ressources

- un programme source .C
- des fichiers .H
- un fichier ressources .RC
- compilateur de ressources RC génère des fichiers .RES puis .EXE
- compilateur CL génère .OBJ
- éditeur de liens LINK4 construit .EXE en utilisant le fichier .DEF et des bibliothèques .LIB

Remarque:

- **Un fichier des ressources contient la description des éléments qui sont susceptibles de changer d'une version de l'application à l'autre: menus, icônes, curseurs, accélérateurs, ...**

Processus d'élaboration:



L'édition de liens dynamique

Lors de l'édition de liens statique, le code externe, se trouvant dans une bibliothèque est chargé en mémoire, lié avec l'application et placé dans le fichier .EXE. Il y a donc duplication .

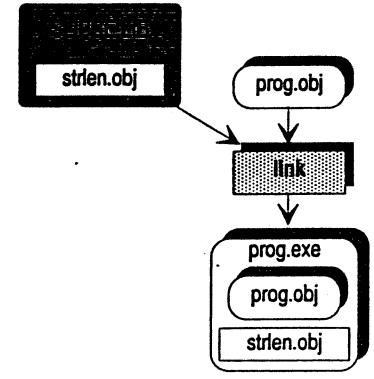
L'édition de liens dynamique ne place pas le code dans l'application, mais y met seulement une référence. Lors de l'exécution, l'appel de la fonction dont l'adresse n'est pas résolue provoque le chargement de la bibliothèque le contenant, si elle n'est pas déjà chargée et la résolution dynamique des adresses avant d'effectuer l'appel.

Ce mécanisme offre plusieurs avantages:

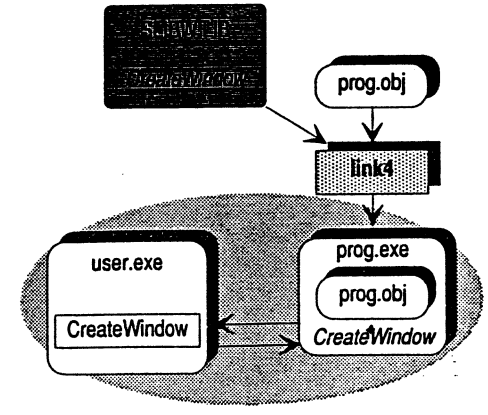
- Les fichiers .EXE sont moins gros
- Le code des bibliothèques est partagé entre applications, et n'est chargé qu'une seule fois en mémoire (multitâche)

- Le système peut évoluer sans que les applications aient à être modifiées.

LINK STATIQUE



LINK DYNAMIQUE

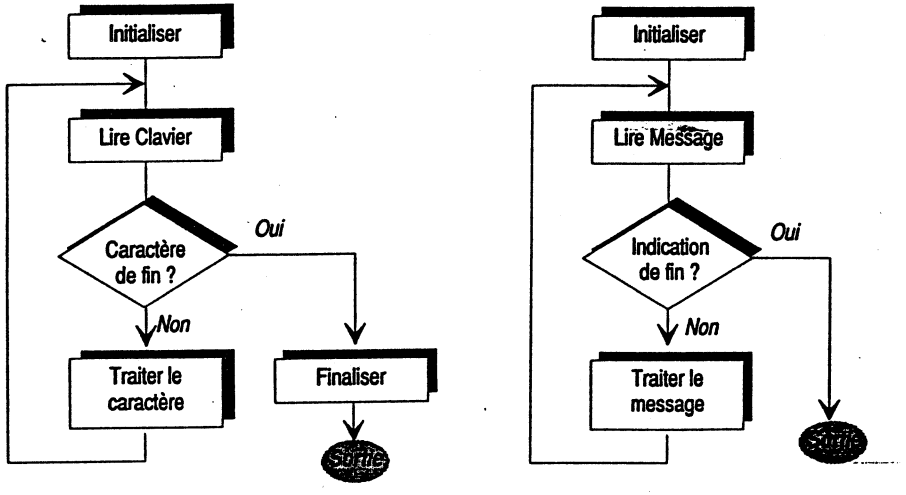


Structure du programme

Structure d'un programme DOS:

- structure linéaire avec un seul point d'entrée

Programme DOS Programme Windows



Structure d'un programme Windows

- la lecture et le traitement ne concernent plus un caractère mais un message.

- ce message peut provenir de différentes sources: souris, clavier, horloge système, noyau Windows ou une autre application (multitâche).

- une application Windows possède en général plusieurs points d'entrée, c'est-à-dire fonctions pouvant être appelées par le système Windows.

- Le traitement d'un message est fait habituellement par l'appel d'une fonction Windows (DispatchMessage) qui appelle la partie de l'application destinataire du message.

- Les deux ingrédiaents nécessaires à la constitution d'une application sont des fenêtres et des messages. Tout affichage à l'écran ou toute saisie de données sont réalisés à l'aide de la fenêtre.

- Une application Windows estessentiellement composée de fenêtres, d'un arbre de fenêtres (fenêtre principale et fenêtres filles).

- A chaque fenêtre est associée une fonction chargée de la gérer: la Windows Procedure ou WndProc, également appelée Dialog Procedure ou DlgProc lorsque la fenêtre est une Boîte de dialogue.

- Une application Windows comprend plusieurs points d'entrée:
  - + un point d'entrée principal: WinMain
  - + des points d'entrée pour chaque fenêtre de l'application: WndProc ou DlgProc
  - + et d'autres: callback functions

Messages

Une application réagit à des événements, représentés par des messages

Un message est constitué par:

+ une identification (plus de 140 messages)
+ des paramètres
+ un destinataire
+ une date et heure d'émission

Une fenêtre ou plus précisément WndProc reçoit des messages et les traite.

Processus de distribution de messages:

Le corps principal de l'application WinMain, tient en général le rôle de facteur de l'application, il est chargé de distribuer les messages.

L'application regarde dans sa liste d'attente si un message est disponible et le retire (fonctions PeekMessage ou GetMessage). Elle le distribue alors à la fenêtre destinataire, représentée par sa WndProc (fonction DispatchMessage).

Le plus petit programme

Sous Dos:

Fichier: PROG.C

```
main (argc, agrv)
int     argc,       /* Nombre d'arguments */
char    *argv[];    /* Pointeurs sur arguments */
{
}
```

Sous Windows:

Fichier: WINMAIN.C

```
/*    WinMain.C
 *    Point d'entrée de l'Application
 */

#include "Windows.h"

/*--------------------WinMain--------------------*
/

int PASCAL WinMain (hInstance, hPrevInstance,
                    lpCmdLine, nCmdShow)
HANDLE  hInstance,      /* Instance courante */
        hPrevInstance;/* Instance précédente */
LPSTR   lpCmdLine;      /* Ligne de commande */
int     nCmdShow;       /* Mode visualisation fenêtre */
{
     return 0;
}
```

Fichier: MINI.DEF

```
NAME mini
DESCRIPTION 'Application Mini'

STACKSIZE       4096
HEAPSIZE        4096
```

Fichier: MINI

```
#-----------------Application Mini---------------
#--------Compilation

Winmain.obj:    Winmain.c
        cl -c -u -W2 -AS - Gsw -Od -Zpe Winmain.c

#------Edition de liens

Mini.exe: Winmain.obj Mini.def
        link4 /a:16 Winmain.obj, Mini, nul, slibw, Mini.def
```

Remarques:

- le fichier WINDOWS.H contient la définition des symboles propres à Windows et les déclarations de fonctions. Il doit être systématiquement inclus dans toute application.
- Le point d'entrée principal est WinMain, il retourne un entier comme code de retour de l'application.
- Les quatre arguments sont
  - + hInstance: le handle de l'instance courante
  - + hPrevInstance: le hande de l'instance précédente
  - + lpCmdLine: pointeur sur la ligne de commande émise
  - + nCmdShow: nombre précusant la façon d'ouvrir la fenêtre pr.
- Le fichier MINI.DEF contient desinformations pour l'éditeur de liens dynamique, entre autres la taille de la pile et du heap.
- Le fichier MAKE MINI permet d'effectuer une action conditionnellement, selon les dates de dernière mise à jour des fichiers.

Une application modèle

Les fichiers sources:

- un ou plusieurs fichiers programme en C, Pascal ou Assembleur: .C, .PAS et .ASM
- un fichier de définition des ressources: .RC
- un fichier de définitions pour l'éditeur de liens: .DEF

Les bibliothèques:

- les bibliothèques utilisées dépendent du compilateur, du modèle mémoire, de la bibliothèque mathématique. Par exemple: SLIBCE.LIB pour MS-C V5.X et modèle small.
- les fonctions Windows s'appuyent sur l'édition de liens dynamique. Les bibliothèques utilisées sont mLIBW.LIB pour une application normale, où m est une lettre dépendant du modèle mémoire (S pour small).

Entités fonctionnelles et fichiers sources:

- Une application Windows peut être découpée et plusieurs entités fonctionnelles:
  - + le corps principal du programme avec la fonction WinMain, qui est unique
  - + chaque ensemble de fonctions associées à une fenêtre principale, comprenant une WndProc et d'autres fonctions liées
- Chaque entité fonctionnelle est implémentée das un module source qui peut ne comprendre qu'un seul fichier source, si celui-ci n'est pas trop gros et écrit dans un langage unique.

Portée des fonctions

Une fonction, définie dans un module source, peut avoir trois niveaux de portée:

1° niveau: la fonction n'est connue que dans le fichier source où elle est définie. Elle est déclarée static:

```
static int Calcul (int n1, int n2)
{.....}
```

Cette fonction ne peut être appelée qu'à partir d'une fonction contenue dans le même source. Elle est locale.

2° niveau: la fonction peut être appelée à partir d'une fonction définie dans un autre fichier source. Elle est déclarée extrn (ou n'a pas de déclaration, car c'est l'option par défaut):

```
void RemiseAZero (BOOL bIndic)
{.....}
```

BOOL est synonyme de int. Il s'agit de fonctions globales à l'application.

3° niveau: la fonction peut être appelée par Windows: elle est un point d'entrée de l'application. Elle est déclarée FAR PASCAL:

```
long FAR PASCAL MainWndProc (hWnd, message, wParam, lParam)
        HWND    hwnd;
        unsigned message;
        WORD    wParam;
        LONG    lParam;
        {.....}
```

Il s'agit d'un appel long, intersegment, avec la convention d'appel de PASCAL: les paramètres sont placés sur la pile de gauche à droite (en nombre fixe) et la restauration de la pile est de la responsabilité de l'appelé.

Les ressources:

- La notion de ressource est un concept nouveau de Windows. Il s'agit d'un fichier d'extension .RES, contenant des données liées à une application. Ce fichier est intégré dans l'exécutable .EXE et fait partie du nouveau format EXE.

- Le fichier .RES est un fichier binaire, résultat de la compilation d'un source, .RC, par un compilateur de ressources RC.

- Le fichier des ressources contient des ressources telles que: les messages (StringTable), Menus, Boites dde dialogue (Dialog), Accélérateurs, curseur, Icônes, polices de caractères,...

- Des outils logiciels spécifiques permettent de créer certaines resources:
  + ICONEDIT pour des Curseurs, Icônes et Images binaires
  + FONTEDIT pour des Polices de caractères
  + DIALOG pour des Boites de Dialogue

Les définitions:

- Le fichier .DEF est un fichier de parmètres lus par l'éditeur de liens LINK4. Il contient des définitions liées au chargement du modèle exécutable et à l'édition de liens dynamique.

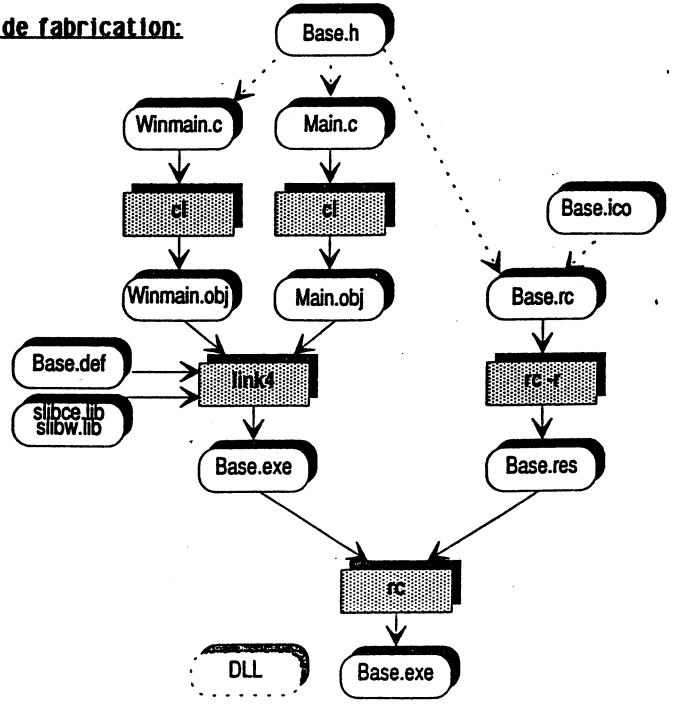
Un modèle

Applicationn appelée BASE, extrêmement simple, qui met en oeuvre les mécanismes essentiels de Windows: elle ouvre une fenêtre et réagit à quelques messages.

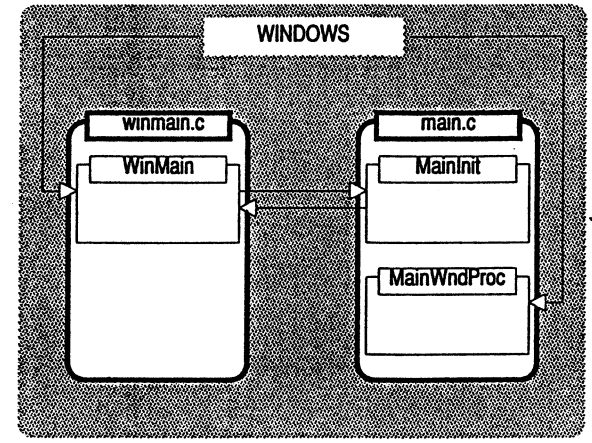
Structure de l'application:

- BASE comprend deux entités fonctionnelles:
  - + WinMain le point d'entrée pricipal et corps de l'application
  - + Main le module de gestion de la fenêtre
- A chacune de ces entités fonctionnelles correspond un fichier source dont le contenu est le code chargé de la gérer: WINMAIN.C et MAIN.C

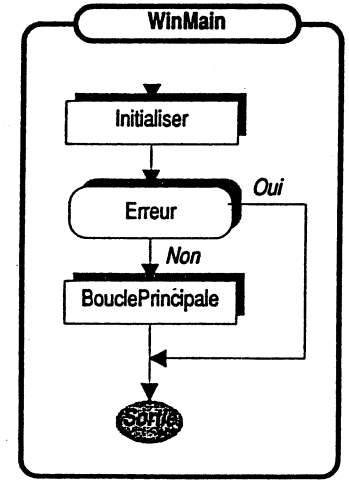
La chaîne de fabrication:



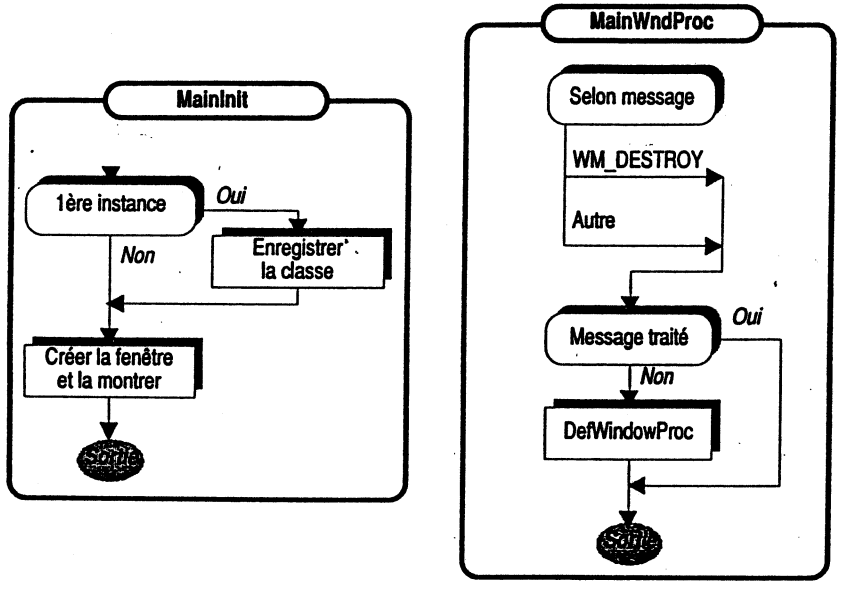
Relations entre modules et Windows:



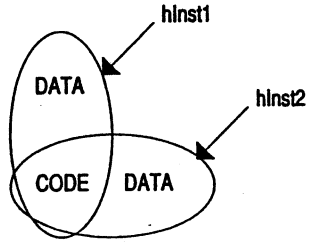
Le module principal: WINMAIN.C



Gestion de la Fenêtre principale: MAIN.C



Code et Données pour deux instances d'une même application:



Le fichier MAKE:

```
APPLI=base                          # Nom de l'application
OBJLISTE=Winmain.obj Main.obj
RCLISTE=$(APPLI).rc $(APPLI).ico Main.h

#---- Options de compilation

#   Option  Signification
#---------------------------
#   c       compilation seule
#   u       supprime les macros prédéfinies
#   W2      niveau de warnings - peut être changé
#   AS      modèle small avec SS != DS - peut être changé
#   Gs      pas de contrôle de débordement de pile (stack probes)
#   Gw      ajoute prologue et épilogue Windows à chaque fonction
#   Od      Pas d'optimisation (facilite debug) - peut être changé
#   Zp      les structures sont packées
#   Ze      autorise les mots clés pascal, far et near
#   Zd      ajoute informations pour symdeb - peut être supprimé
#   D       Définit le nom du .h principal

COPT=-c -u -W2 -AS -Gsw -Od -Zped -D APPLIH=\"$(APPLI).h\"

#---- Options de link4

#   Option  Signification
#---------------------------
#   a:16    alignement des segments data sur une frontière de 16 bytes
#   map     ajouter des infos symboliques pour symdeb - peut être supprimé
#   li      ajouter des infos sur lignes pour symdeb - peut être supprimé

LNKOPT=/a:16/map/li
#---- Règle d'inférence
.c.obj:
    cl $(COPT) $*.c

#---- Compilations

Winmain.obj: $*.c $(APPLI).h
Main.obj:    $*.c $*.h $(APPLI).h

$(APPLI).res: $(RCLISTE)
    rc -r $(APPLI).rc

#---- Edition de liens

$(APPLI).lnk: $(APPLI)
    echo $(LNKOPT) $(OBJLISTE)>$(APPLI).lnk
    echo $(APPLI)>>$(APPLI).lnk
    echo $(APPLI)>>$(APPLI).lnk
    echo slibw>>$(APPLI).lnk
    echo $(APPLI).def>>$(APPLI).lnk

$(APPLI).exe: $(OBJLISTE) $(APPLI).def
    link4 @$(APPLI).lnk
    rc $(APPLI).res
    mapsym $(APPLI)

$(APPLI).exe: $(APPLI).res
    rc $(APPLI).res
```

Source: WINMAIN.C

```
/*
 *      WinMain.c
 *      Point d'entrée principal de l'Application
 */

#include "Windows.h"

#include APPLIH

/* ------------------------ Initialiser ------------------------ */

static BOOL Initialiser (nCmdShow)
int             nCmdShow;       /* Mode visualisation fenêtre */
{
BOOL            bResultat = FALSE;  /* Valeur retour de la fonction */
/* ==== Initialisation fenêtre principale */
    bResultat = MainInit (nCmdShow);
    return bResultat;
}

/* ---------------------- BouclePrincipale ---------------------- */

static int BouclePrincipale ()
{
MSG             Msg;                /* Structure messages */

     while (GetMessage ((LPMSG)&Msg, NULL, 0, 0))
    {
        TranslateMessage ((LPMSG)&Msg);
        DispatchMessage  ((LPMSG)&Msg);
    }

     return Msg.wParam;
 }

 /* -------------------------- WinMain -------------------------- */

 int PASCAL WinMain (hInstance, hPrevInstance, lpCmdLine, nCmdShow)
 HANDLE          hInstance,          /* Instance courante */
                 hPrevInstance;      /* Instance précédente */
 LPSTR           lpCmdLine;          /* Ligne de commande */
 int             nCmdShow;           /* Mode visualisation fenêtre */
 {
 int             nResultat = 0;      /* Valeur de retour */
 /* ==== Place les instances dans les variables globales */

     hInst = hInstance;              /* Instance Courante */
     hPrevInst = hPrevInstance;      /* Instance Précédente */
 /* ==== Initialise et lance la boucle principale */

      if (Initialiser (nCmdShow))
         nResultat = BouclePrincipale ();    /* Boucle sur messages */
    return nResultat;
 }
```


Source: MAIN.C

```
/*
 *      Main.c
 *      Gestion de la fenêtre principale
 */
#include "windows.h"
#include APPLIH
   #include "main.h"

/* ------------------------ MainWndProc ------------------------ */

long FAR PASCAL MainWndProc (hWnd, wMsg, wParam, lParam)
HWND            hWnd;
unsigned        wMsg;
WORD            wParam;
LONG            lParam;
{
long            lResultat = 0L;     /* Valeur de retour */
BOOL            bTraite = TRUE;     /* Indique commande traitée */

  switch (wMsg)
    {
    case WM_DESTROY:                /* Fin d'appli */
        PostQuitMessage (0);
        break;
    default:
        bTraite = FALSE;
        break;
    }
    if (!bTraite)

        lResultat = DefWindowProc (hWnd, wMsg, wParam, lParam);
    return lResultat;
}
```

Source: MAIN.C (suite)

```
/* ---------------------------- MainInit ---------------------------- */

BOOL MainInit (nCmdShow)
int             nCmdShow;           /* Mode visualisation fenêtre */
{
BOOL            bResultat = FALSE;  /* Valeur retour de la fonction */
WNDCLASS        Class;              /* Structure classe fenêtre */
HWND            hWnd;               /* Handle fenêtre */
char            Titre [50];         /* Titre fenêtre */
    do
    {
/* ==== Enregistre la classe de la fenêtre */

        if (!hPrevInst)
        {
            Class.hCursor       = LoadCursor (NULL, IDC_ARROW);
            Class.hIcon         = LoadIcon (hInst, (LPSTR)MAIN_ICON);
            Class.lpszMenuName  = (LPSTR)NULL;
            Class.lpszClassName = (LPSTR)MAIN_CLASS;
            Class.hbrBackground = (HBRUSH)GetStockObject (WHITE_BRUSH);
            Class.hInstance     = hInst;
            Class.style         = CS_HREDRAW | CS_VREDRAW;
            Class.lpfnWndProc   = MainWndProc;
            Class.cbClsExtra    = 0;
            Class.cbWndExtra    = 0;
            if (!(bResultat = RegisterClass ((LPWNDCLASS)&Class)))
                break;
        }

    LoadString (hInst, MAIN_TITRE, (LPSTR)Titre, sizeof (Titre));

    hWnd = CreateWindow ((LPSTR)MAIN_CLASS, (LPSTR)Titre,
            WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,

                (HWND)NULL, (HMENU)NULL, (HANDLE)hInst, (LPSTR)NULL);
            if (!hWnd)
                break;                  /* Pas créée */

        ShowWindow (hWnd, nCmdShow);
            UpdateWindow (hWnd);
            bResultat = TRUE;
        }   while (0);
        return bResultat;
    }
```


Fichier inclus: BASE.H

```
/* --------------- Variables globales à l'application -------------- */

 HANDLE          hInst;              /* Instance de l'application */
 HANDLE          hPrevInst;          /* Instance précédente */

 #define MAIN_ICON       "Icon"
 #define MAIN_ICON_RC    Icon
 #define MAIN_CLASS      "Base"
 #define MAIN_TITRE      1
 /* ==== Déclaration de fonctions */

 BOOL            MainInit (int);     /* MAIN.C */
```

Fichier de ressources: BASE.RC

```
 #include "style.h"
#include "base.h"

 MAIN_ICON_RC    Icon    base.ico

 StringTable
{

    MAIN_TITRE, "Application Windows de base"
 }
```

Fichier de définitions: BASE.DEF

```
 NAME
DESCRIPTION 'Modèle d'application de base'

STUB    'WINSTUB.EXE'


CODE    MOVEABLE
DATA    MOVEABLE MULTIPLE

HEAPSIZE  4096
STACKSIZE 4096

EXPORTS
    MainWndProc         @1
```

MS - WINDOWS II
Messages

Les messages sont le moyen privilégié, pour une fenêtre ou une application, de communiquer avec l'extérieur.

- Une application est constituée de fenêtres. Chaque fenêtre et sa fonction WndProc associée forment une entité relativement autonome.

Les messages sont transmis d'un émetteur vers un destinataire selon deux schémas possibles:

- à travers une file de messages,
- par appel direct

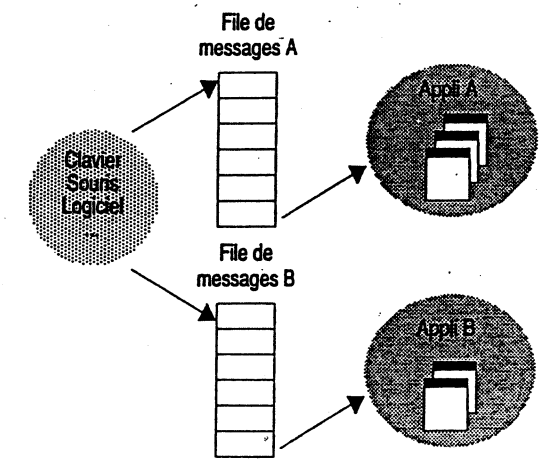
File de messages:

A chaque instance d'une application est associée une file d'attente de type FIFO (Premier entré , premier sorti).

Un message peut avoir pour origine un périphérique ou un logiciel. Dans les deux cas Windows place le message dans la file (avec ou sans interruption matérielle).

Une application place un message dans une file de message en utilisant les fonctions:

- PostMessage ou PostQuitMessage si le destinataire est une fenêtre
- PostAppMessage si le destinataire est une application

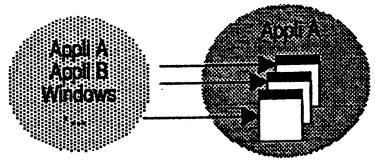


Envoi direct:

L'envoi direct à son destinataire sans passer par la file de messages:

- l'appelant n'a pas à connaître le nom de la fonction gérant la fenêtre destinataire mais uniquement le handle de la fenêtre.
- la fonction réceptrice reçoit le message de la même façon que si celui-ci ait été transmis par la file de messages ou directement.

L'envoi direct d'un message vers une ou plusieurs fenêtres se fait en utilisant la fonction SendMessage.



Distribution des messages:

La lecture d'un message dans la file d'attente se fait en utilisant une des fonctions GetMessage (bloquante) ou PeekMessage (non bloquante).

Principe de circulation d'un message:

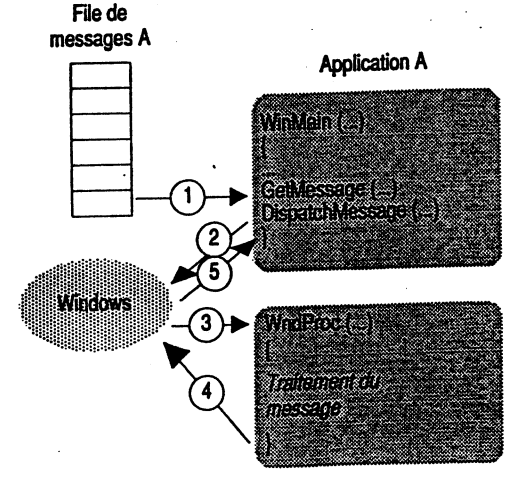
1/ Lecture par GetMessage ou PeekMessage

2/ Distribution: envoi du message à la fenêtre par DispatchMessage, pour le redonner à Windows, lequel se charge de la distribution.

3/ Appel de la fonction destinataire. Il s'agit d'un appel direct par Windows, de la fonction, avec le message comme paramètre. La fonction traite le message.

4/ Retour de la fonction destinataire. La fonction de traitement du message redonne le contrôle à Windows, avec un code retour.

5/ Retour de distribution. Windows retourne de DispachMessage, avec le code retour de la fonction destinataire. L'application habituellement boucle sur l'étape 1, c'est-à-dire GetMessage.



Structure du message:

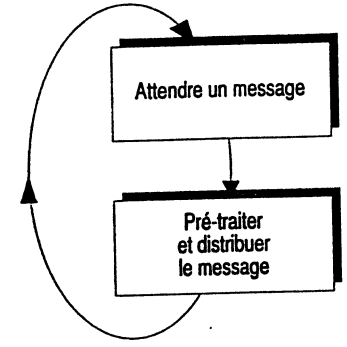
```
typedef struct
    {
    HWND    hwnd;     handle de la fenêtre destinataire
    WORD    message;  message identifié par son n° de code
    WORD    wParam;   paramètre court
    LONG    lParam;   paramètre long
    DWORD   time;     heure
    POINT   pt;       coordonnées du curseur
    } MSG;
```

Le message reçu par son destinataire WndProc

Seuls les quatre premiers éléments sont transmis, les autres peuvent être obtenus grâce aux fonctions GetMessageTime et GetMessagePos.

Les messages sont obtenus et traités à deux niveaux:

- dans le boucle principale, où se fait la lecture de la file,
- dans une WndProc, où s'effectue l'exploitation d'un message.



Boucle principale:

La boucle principale de l'application a une double fonction:

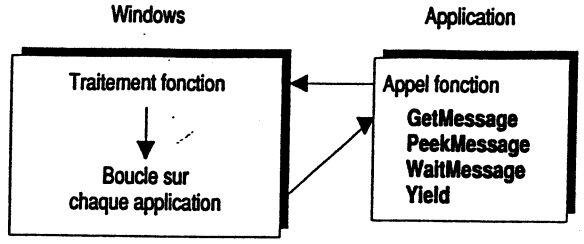
- obtenir un message, ou effectuer toute autre action en atttendant la disponibilité d'un message,
- effectuer un prétraitement du message lu, puis le distribuer à la WndProc chargée de son traitement effectif.

Lecture de la file de messages:

Objectif: obtenir un message, mais aussi donner la main aux autres applications car Windows est un multitâche non préemptif (pas d'interruption de tâches).

- Il appartient donc à chaque application de redonner le contrôle à Windows le plus souvent possible.

Quatre fonctions permettent à une tâche de redonner le contrôle à Windows:



GetMessage	Lit le prochain message. Possibilité de filtre. Bloquant.
PeekMessage	Voit s'il y a un message dans la file. Possibilité de filtre. Non bloquant.
WaitMessage	S'endort, jusqu'à la disponibilité d'un message. Pas de filtre. Bloquant.
Yield	Donne la main aux autres applications (un tour). Non bloquant.
Set MessageQueue	Modifie la taille de la file de messages (8 messages par défaut)

Les deux fonctions fondamentales sont GetMessage et PeekMessage.

GetMessage: bloquante vis-à-vis de l'application, donne la main à WINDOWS s'il n'y a pas de message en attente.

BOOL GetMessage (lpMsg, hWnd, wMsgFiterMin, wMsgFilterMax)

Lit le prochain message de la file destiné à la fenêtre de handle hWnd (ou toutes les fenêtres de l'instance, si hWnd nul) de valeur comprise entre les bornes du filtre, stocke le message à l'adresse lpMsg et retourne FALSE si fin d'application (message WM_QUIT) sinon TRUE.

Filtres sur les messages lus peuvent concerner

- la fenêtre destinataire (fenêtre particulière si hWnd ≠NULL) ou toutes les fenêtres de l'application
- filtre sur certains messages: par exemple en provenance du clavier ou de la souris

Fin du programme:

GetMessage retourne une valeur fausse (0). Ceci correspond au traitement de WM_QUIT

PeekMessage: fonctionnalités semblables à GetMessage sauf que cette fonction ne retire pas le message de la file et n'est pas bloquante vis-à-vis de l'application appelante. Elle voit "à la volée" s'il y a un message en attente.

BOOL PeekMessage (lpMsg, hWnd, wMsgFiterMin,
wMsgFilterMax, bRemoveMsg)

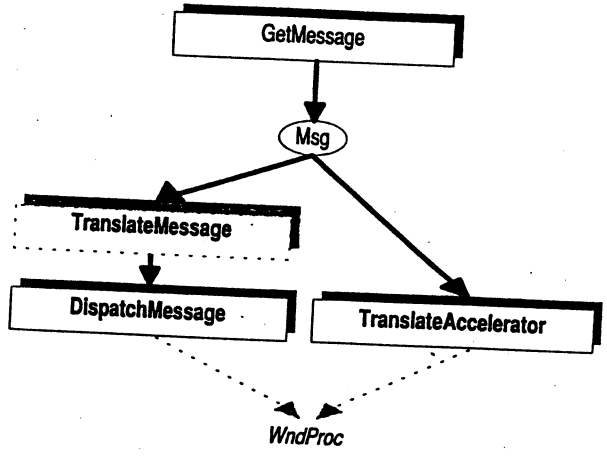
- Lit le prochain message de la file, s'il y en a un (mêmes conditions de fitre que GetMessage). Stock éventuellement le message à l'adresse lpMsg. Retire le message de la file, si bRemoveMsg est TRUE. Retourne TRUE s'il y a un message dans la file, sinon FALSE.

Pourquoi PeekMessage? Pour prendre en compte des événements asynchrones.

Prétraitement et distribution des messages

- GetMessage et PeekMessage sont dans la boucle principale de WinMain.
- La prochaine étape est la lecture d'un message et donc sa distribution à la fenêtre destinataire, à l'aide de la fonction DispatchMessage et des fonctions de prétraitement standard de messages relatifs au clavier:

  TranslateMessage et TranslateAccelerator.

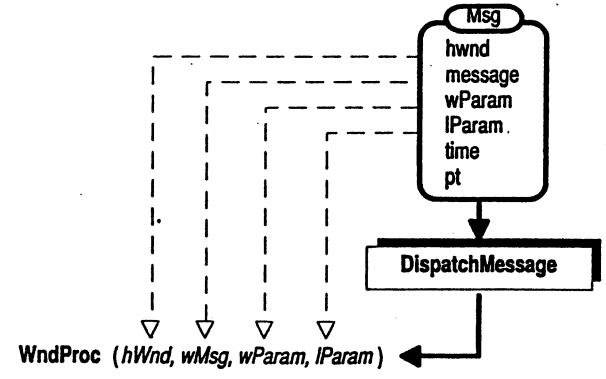


DispatchMessage: cette fonction distribue un message à une fonction destinataire. Elle réalise donc l'appel de la fonction (WndProc) associée à la fenêtre destinataire du message.

```
long DispatchMessage (lpMsg)
```

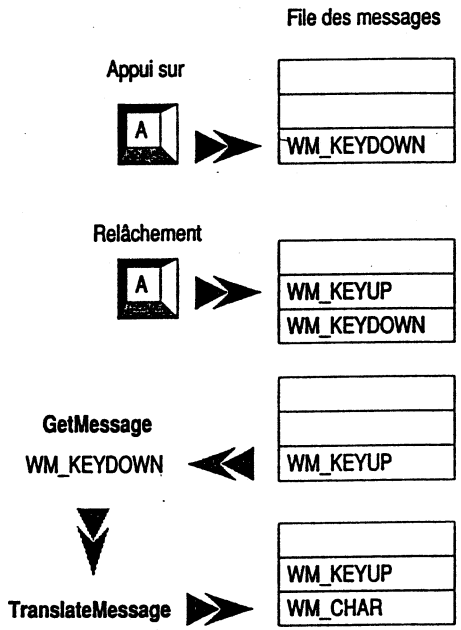
Exemple:

```
static int BouclePricipale ()
{
int     nResultat;          /* Valeur de retour de la fonction */
MSG     Msg;                /* Structure messages */
        while (GetMessage(&Msg,NULL, 0,0))
        {
           TranslateMessage.(&Msg);
           DispatchMessage  (&Msg);
        }
        return Msg.wParam
}
```



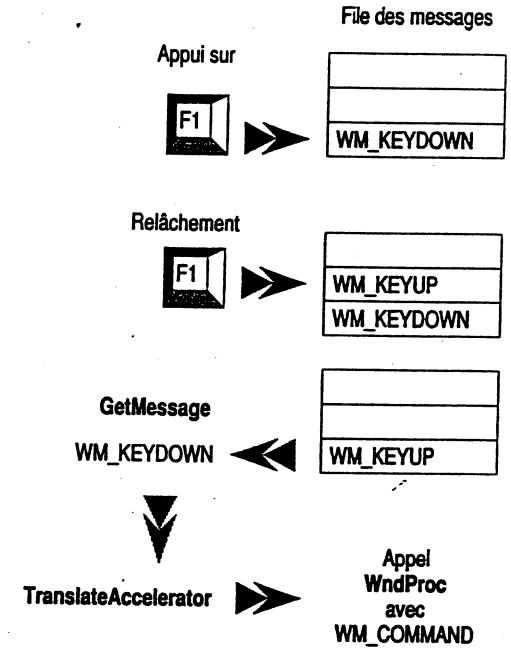
TranlateMessage: génère un message caractère à partir de message touches virtuelles.

bool TranslateMessage (lpMsg)



TranslateAccelerator: Traite les touches accélératrices à l'aide de la table et le distribue. Elle prend donc la place de **DispatchMessage** qui ne doit pas être appelée dans ce cas.

bool TranslateAccelerator (hWnd, hAccTable, lpMsg)



Emission de messages:

Pourquoi et à qui émettre les messages?

- vers des applications: véhicule de protocoles de communication interapplication sous Windows
- vers des fenêtres: remplace un appel de fonction

Un message émis est une structure de type MSG: hwnd, message, wParam et lParam. Windows complète avec time et pt.

Fonctions d'émission:

PostMessage
Place un message destiné à une fenêtre dans la file d'une application

PostAppMessage
Place un message dans la file d'une application

PostQuit
Place un message WM_QUIT dans la file de l'application

SendMessage
Emet un message directement à une fenêtre

SendDlgItemMessage
Emet un message directement à une fenêtre de contrôle.

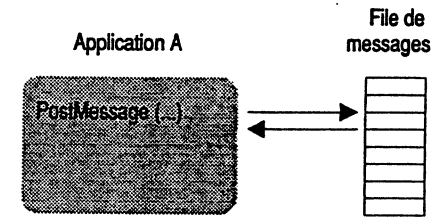
ReplyMessage
Acquitte un message envoyé par SendMessage

Remarques:

- Les fonctions "Post" sont asynchrones, "Send" sont synchrones (bloquantes pour l'appelant): elles fonctionnent comme un appel de fonction.
- Les fonctions principales sont PostMessage et SendMessage.
- SendDlgItemMessage est une combinaison de GetDlgItem et SendMessage

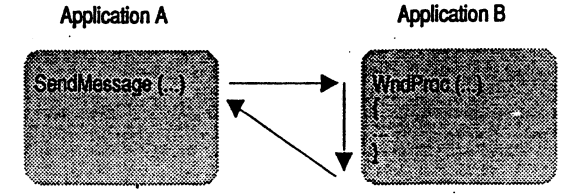
PostMessage: permet d'envoyer un message vers une fenêtre précise ou vers toutes les fenêtres du système (si hWnd = 0xFFFF). Windows fait lors une copie du message pour chaque fenêtre.

BOOL PostMessage (hWnd, wMsg, wParam, lParam)



SendMessage: permet d'envoyer immédiatement un message à une ou plusieurs fenêtres et d'attendre la réponse. Particulièrement utilisé pour transmettre des instructions à des fenêtres contrôle.

long SendMessage (hWnd, wMsg, wParam, lParam)



Traitemnent des messages: la WndProc

- La majorité des messages arrivent à la WndProc destinataire, celle dont le handle est le hvnd de la structure MSG (via file d'attente: PostMessage ou directement: SendMessage).

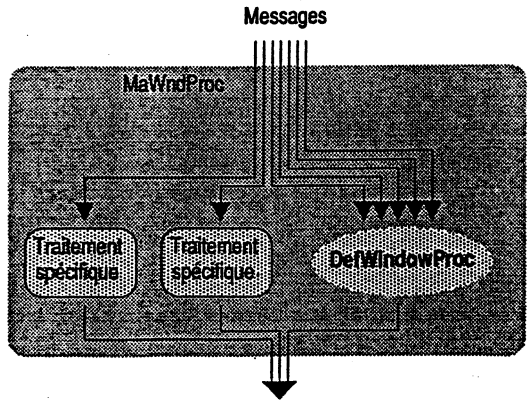
- Les quatre paramètres de WndProc sont hWnd, wMsg, wParam et lParam.

Pour traiter correctement des messages, il faut savoir que:

- Windows envoie de nombreux messages (déplacement du curseur conduit à émettre le message vers la WndProc: nécessite donc une vitesse d'échantillonnage importante)

- une application ne traite que les messages qui l'intéressent. Tous les autres (la majorité) sont transmis à une fonction de la bibliothèque Window: DefWindowProc: qui effectue le traitement correct par défaut.

- Le code retour de la WndProc est utilisé par Windows (qui est l'appelant), il est donc important.



Exemple de programme:

```
#include "windows.h"
#include APPLIH
#include "main.h"

static BOOL Create (hWnd, lParam)
HWND            hWnd;
DWORD           lParam;
{
    return FALSE;
}

static BOOL Move (hWnd, lParam)
HWND            hWnd;
DWORD           lParam;
{
    return FALSE;
}

static BOOL Mouse (hWnd, wMsg, wParam, lParam)
HWND            hWnd;
WORD            wMsg;
WORD            wParam;
DWORD           lParam;
{
    return FALSE;
}

long FAR PASCAL MainWndProc (hWnd, wMsg, wParam, lParam)

HWND            hWnd;
WORD            wMsg;
WORD            wParam;
DWORD           lParam;
{

long            lResultat = 0L;     /* Valeur de retour */
BOOL            bTraite = TRUE;     /* Indique commande traitée */

    switch (wMsg)
    {
    case WM_CREATE:             /* Création de la fenêtre */
        bTraite = Create (hWnd, lParam);
        break;
    case WM_DESTROY:            /* Fin d'application */
        PostQuitMessage (0);
        break;
    case WM_MOVE:               /* Déplacement de la fenêtre */
        bTraite = Move (hWnd, lParam);
        break;
    case WM_SIZE:               /* Changement de taille de la
                                   fenêtre */
        bTraite = Size (hWnd, wParam, lParam);
        break;
```

```
        case WM_TIMER:              /* Message de l'horloge */
            bTraite = Timer (hWnd, wParam);
            break;
        case WM_VSCROLL:            /* Saisie sur la barre
                                       de scroll verticale */
            bTraite = VScroll (hWnd, wParam, lParam);
            break;
        case WM_HSCROLL:            /* Saisie sur la barre
                                       de scroll horizontale */
            bTraite = HScroll (hWnd, wParam, lParam);
            break;
        case WM_MOUSEMOVE:          /* Action sur la souris */
        case WM_LBUTTONDOWN:
        case WM_LBUTTONUP:
        case WM_RBUTTONDOWN:
        case WM_RBUTTONUP:
        case WM_LBUTTONDBLCLK:
        case WM_RBUTTONDBLCLK:
            bTraite = Mouse (hWnd, wMsg, wParam, lParam);
            break;
        default:
            bTraite = FALSE;
            break;
        }
        if (!bTraite)
            lResultat = DefWindowProc (hWnd, wMsg, wParam, lParam);
        return lResultat;
}

BOOL MainInit (nCmdShow)

int             nCmdShow;           /* Mode visualisation fenêtre */

{
BOOL            bResultat = FALSE;  /* Valeur retour de la fonction */

WNDCLASS        Class;              /* Structure classe fenêtre */
HWND            hWnd;               /* Handle fenêtre */
char            Titre [50];         /* Titre fenêtre */
    do
    {

/* ==== Enregistre la classe de la fenêtre */

        if (!hPrevInst)             /* Seulement si 1ère instance */
        {
            Class.hCursor       = LoadCursor (NULL, IDC_ARROW);
            Class.hIcon         = LoadIcon (hInst, MAIN_ICON);
            Class.lpszMenuName  = (LPSTR)NULL;
            Class.lpszClassName = (LPSTR)MAIN_CLASS;
            Class.hbrBackground = (HBRUSH)GetStockObject (WHITE_BRUSH);
            Class.hInstance     = hInst;
            Class.style         = CS_HREDRAW | CS_VREDRAW;
            Class.lpfnWndProc   = MainWndProc;
            Class.cbClsExtra    = 0;
            Class.cbWndExtra    = 0;
            if (!(bResultat = RegisterClass (&Class)))
                break;
        }
```



```
        LoadString (hInst, MAIN_TITRE, Titre, sizeof (Titre));

        hWnd = CreateWindow (MAIN_CLASS, Titre,
            WS_OVERLAPPEDWINDOW,
            CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
            NULL, NULL, hInst, NULL);

        if (!hWnd)
            break;                  /* Pas créée */


        ShowWindow (hWnd, nCmdShow);
        UpdateWindow (hWnd);
        bResultat = TRUE;

    }   while (0);
    return bResultat;
}
```

MS - WINDOWS II
Fenêtres

La fenêtre est un objet avec un cycle de vie: naissance (création), période de vie active (évolution, chagement de forme, d'apparence,...) et mort (destruction).

- Les enfants d'une fenêtre ne survivent pas à sa mort.
- Une fenêtre n'a pas nécessairement de parent fenêtre, mais elle est créée à l'initiative de son propriétaire: une instance d'application.
- une fenêtre possède un certain nombre d'attributs qui déterminent son apparence et son comportement. Ceux-ci sont fixés, en partie, par la classe à laquelle elle appartient.

Une fenêtre, deux rôles:

- visualisation (affichage d'informations)
- communication (transmission de mesages)

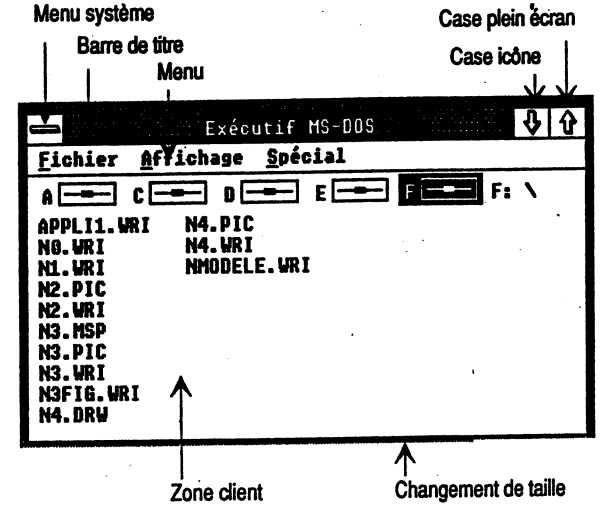
Une fenêtre: un objet, une classe, des attributs, des méthodes

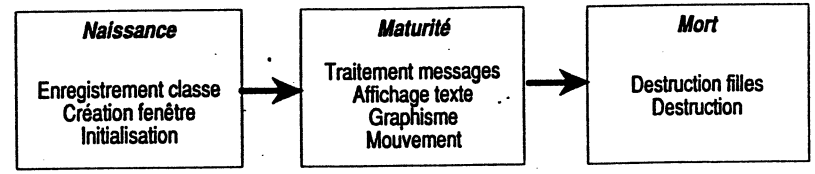
Une fenêtre, deux zones:

- zone client: Windows fournit un grand nombre de fonctions permettant d'y agir: écrire, dessiner, ...

  Toute intervention est transmise à la WndProc associée (déplacement de la souris, clic sur un bouton,...)

- zone hors client: Chaque élément cette zone a une fonction précise et est entièrement géré par Windows.





Naissance d'une fenêtre:

- Plusieurs entités contribuent à la création d'une fenêtre: instance de l'application (propriétaire de la fenêtre), un parent éventuel, et une classe.

- Toute fenêtre fait partie d'une classe (propriétés communes et la WndProc).

- Une classe est désignée par un nom (chaîne de caractères terminée par un nul). Certaines classes sont prédéfinies par Windows, d'autres peuvent être créées par l'application, mais attention: le nom de la classe a une portée globale au système Windows !!!

La structure:

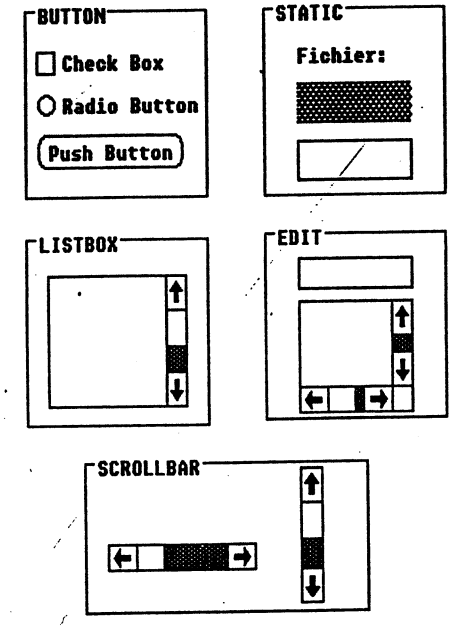
```
typedef struct
{
Word      style;                          style classe
long      (FAR PASCAL *lpfnWndProc) ();   fonction de gestion
int       cbClsExtra;
int       cbWndExtra;
HANDLE    hInstance;                      propriétaire
HICON     hIcon;                          icône de l'appli
HCURSOR   hCursor;                        curseur
HBRUSH    hbrBackground;                  pinceau
LPSTR     lpszMenuName;                   menu
LPSTR     lpszClassName;
}         WNDCLASS;
```

Classes prédéfinies:

- pour les classes prédéfinies le comportement et la gestion sont gérés par les WndProc du noyau. L'application se charge des fonctionnalités spécifiques: initialisation, obtention de données.

- Cinq classes de fenêtres de contrôle (Boites de dialogue) ont été prédéfinies avec des variantes sur le style de la fenêtre:

  BUTTON, STATIC, LISTBOX, EDIT, SCROLLBAR



CLASSES SPECIFIQUES

La création d'une classe s'effectue en deux étapes:

- effectation aux éléments d'une structure WNDCLASS des valeurs définissant la classe,
- enregistrement de celle-ci auprès de Windows en appelant la fonction RegisterClass.

  BOOL RegisterClass(lpWndClass)

  TRUE en cas de succès, FALSE sinon (Pb. place mémoire)

- On peut connaitre l'état d'une classe déjà créée par

  WORD GetClassWord (hWnd, nIndex)
  LONG GetClassWord (hWnd, nIndex)

  et agir sur la définition par

  - Word SetClassWord (hWnd, nIndex, nNewWord)
    Long SetClassWord (hWnd, nIndex, nNewLONG)

  avec des paramètres pour nIndex:
  LONG: GCL_MENUNAME, GCL_WNDPROC
  WORD: GCW_CBCLSEXTRA, GCW_CBWNDEXTRA, ...

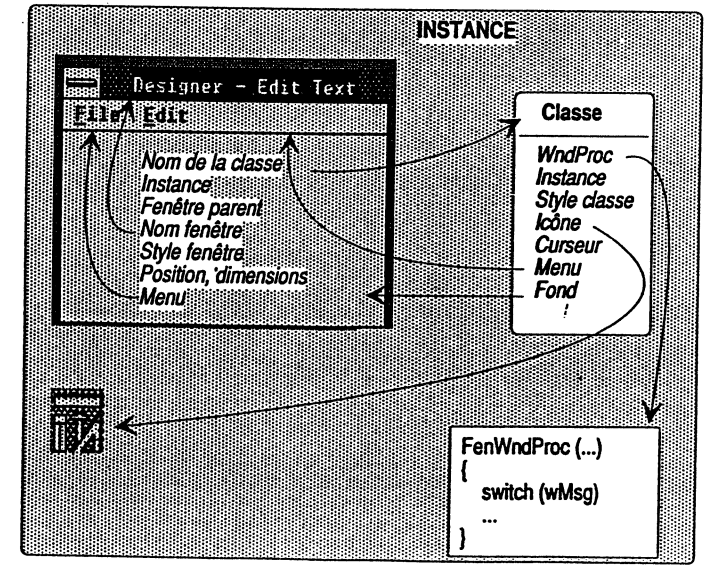
Création de la fenêtre:

- pour créer une fenêtre (d'une classe) on fait appel à la fonction CreateWindow:

  HWND CreateWindow(lpClassName, lpWindowName, dwStyle, X, Y, nWidth, nHeight, hWndParent, hMenu, hInstance, lpParam)

  avec des paramètres de deux sortes:

- paramètres familiaux: nom de la classe, l'instance propriétaire, fenêtre mère
- paramètres propres à la nouvelle fenêtre: nom de la fenêtre, son style, sa position, le menu associé et un paramètre complémentaire.



Les styles de fenêtres (apparence)

- **catégorie:** fenêtre principale (overlapped)
  fenêtre boîte de dialogue ou message
  fenêtre enfant

- **éléments de la zone non client:** bordure, barre de titre,..

- **styles de contrôles:** boutons, listes,...

L'évolution d'une fenêtre est possible grâce aux fonctions: GetWindowXXX, SetWindow,XXX de connaitre les valeurs actuelles (Get) et en mettre d'autres (Set) avec la même syntaxe que pour des classes.

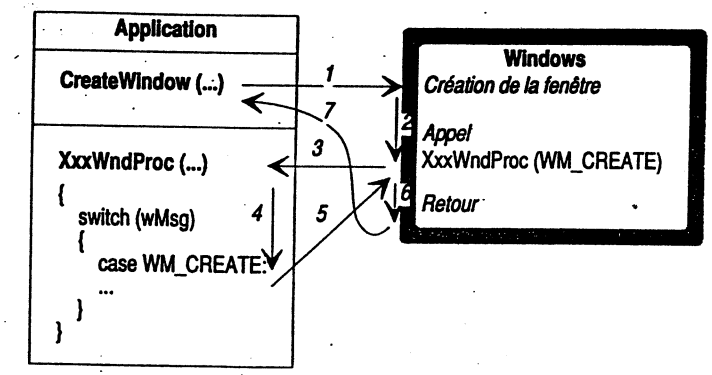
GetWindowsWord (hWnd,nIndex) ou Long
SetWindowsWord (hWnd,nIndex, nNewWord) ou Long

avec des valeurs pour nIndex: GWL_STYLE, GWL_WNDPROC, GWW_INSTANCE, GWW_HWNDPARENT, ...

- Ces fonctions GetXxx et SetXxx sont utilisées généralement pour accéder aux données.

Création d'une fenêtre

A la création (CreateWindow) Windows envoie le message WM_CREATE qui indique que la fenêtre est créée., mais n'est pas encore à l'écran. C'est le moment pour effectuer des initialisations, d'autant plus que lPram pointe sur une structure CREATESTRUCT comprenant les paramètres passés à CreateWindow. On peut alors déplacer la fenêtre avec MoveWindow, changer son titre avec SetWindowText.....



Visualisation de la fenêtre:

Différentes solutions sont possibles:

1/ mettre WS_VISIBLE dans CreateWindows,

2/ Appeler ShowWindow après CreateWindow avec nCmdShow précisant le mode d'affichage,

3/ Appel à UpdateWindow permet de dessiner immédiatement le contenu de la fenêtre.

Mort de la fenêtre:

- destruction explicite: DestroyWindow

- disparition de la fenêtre mère, de l'instance de l'application ou du Windows lui même.

La mort volontaire:

- message WM_CLOSE, envoyé par une fenêtre (le plus souvent FERMER du menu système). Il est interprété par DefWindowProc et conduit à l'appel de la fonction DestroyWindow qui envoie un message WM_DESTROY à la fenêtre et ses filles.

La mort imposée:

- c'est la réception d'un message WM_DESTROY d'un des ancêtres ou WM_QUIT de l'instance de l'application. On a alors juste le temps de fermer des fichiers.

La mort proposée:

- le message WM_QUERYENDSESSION est envoyé à toutes les fenêtres. Chaqune peut répondre qu'elle accepte et c'est ce que fait DefWindowProc. Mais il suffit qu'une seule fenêtre réponde "non" pour que le processus s'arrête et que la session ne soit pas terminée.

Vie quotidienne:

- Recevoir et traiter des messages:

  La WndProc reçoit tous les messages destinés à la fenêtre (soit directement soit par la boucle principales et DispatchMessage)

- La WndProc transmet la majorité des messages à la fonction DefWindowProc, qui les traite de façon standard.

- Composer une application riche en interactions avec l'utilisateur revient , en général, à intercepter et à traiter de façon spécifique un grand nombre de messages.

- Les principales catégories de messages que WndProc peut recevoir:
  + Messages liés à la gestion de fenêtres
  + Messages de saisie de données
  + Messages du système
  + Message du presse-papier
  + Message d'information du système
  + Messages liés aux contrôles
  + Messages de la zone hors client

Exemple d'application:

L'application FILLE fonctionne comme BASE avec deux incéments principaux:

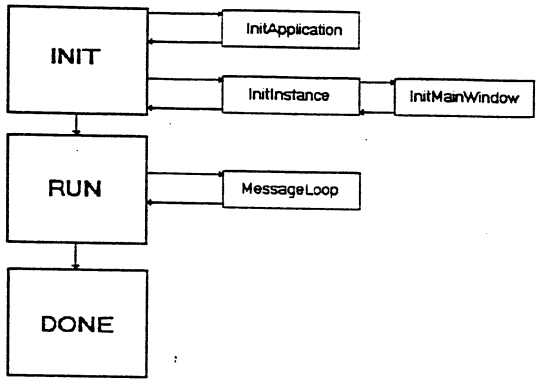
- Une seconde fenêtre est créée immédiatement après la fenêtre principale, en tant que fille de celle-ci.

- Le fenêtre principale affiche en permanence la position et les dimensions de cette seconde fenêtre.

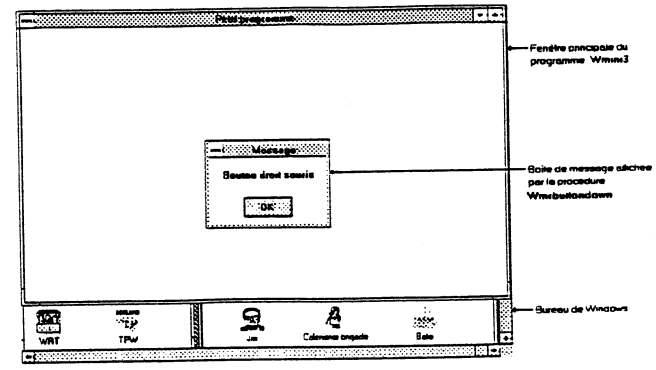
- L'affichage comprend trois lignes: les coordonnées du coin supérieur gauche de Fen par rapport à l'écran sur la primère ligne; les mêmes coordonnées par rapport à la zone client de Main sur la seconde ligne; enfin la largeur et la hauteur de la fenêtre.

- La fenêtre fille, appellée Fen, possède une barre de titre et une bordure épaisse, ce qui permet à l'utilisateur de la déplacer et de modifier ses dimensions. A chacune de ces modifications, Fen envoie un message à sa mère, Main. Celle-ci obtient les dimensions de Fen et les réaffiche.

Turbo-Pascal pour WINDOWS




```
program wmini1;
uses Wobjects;
{ Déclaration des objets }
type
T_application = object(TApplication)
    procedure initmainwindow;virtual;     { Procédure héritée de TApplication }
end;
{ Développement des méthodes }
procedure T_application.initmainwindow;
begin
   { Création de notre propre fenêtre principale }
   mainwindow:= new(pwindow, init(nil, 'Petit programme'));
end;
{ Déclaration des variables globales }
var mon_appli: T_application;            { mon_appli est une instance de TApplication }
{ Début programme principal }
begin
   mon_appli.init('Wmini1');
   mon_appli.run;
   mon_appli.done;
end.
```

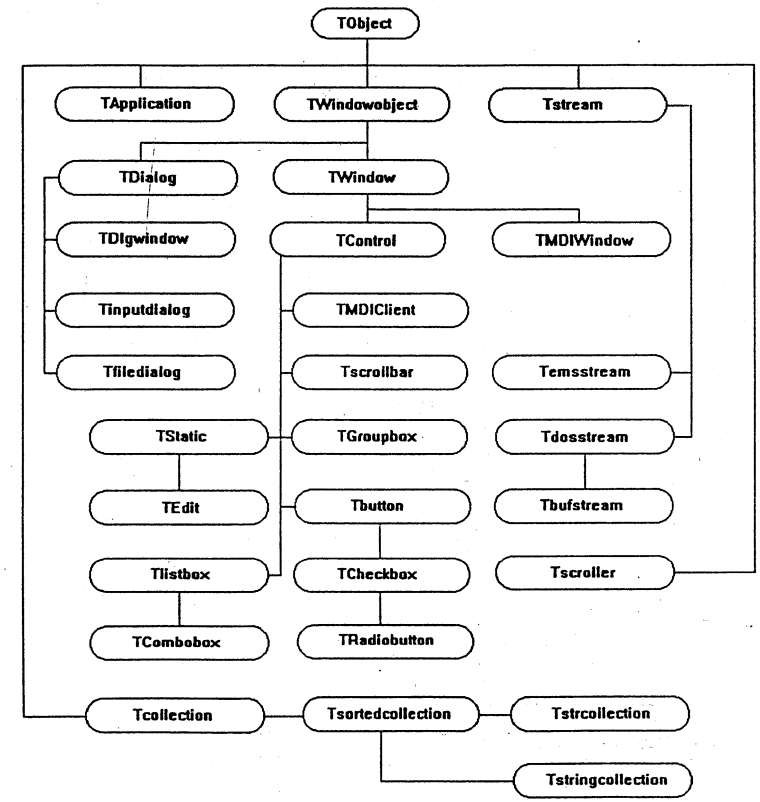


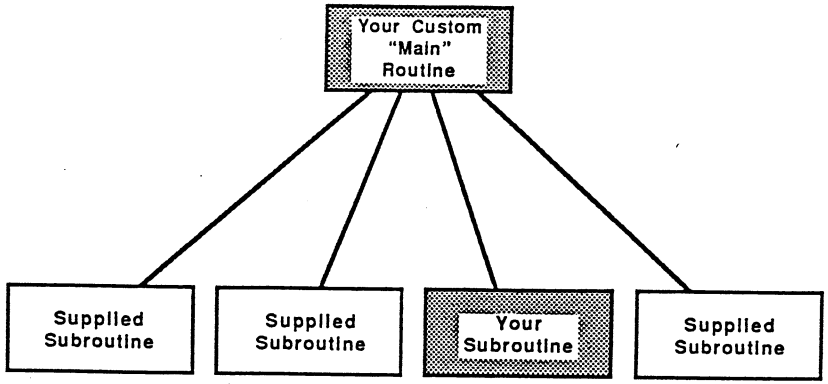

```
program wmini3;
uses Wintypes, Winprocs, Wobjects;
type                                                    { Déclaration des objets }
T_application = object(TApplication)
    procedure initmainwindow;virtual;
end;
P_fenetre = ^T_fenetre;
T_fenetre = object(TWindow)
   { wmlbuttondown sera exécutée si on appuie sur le bouton gauche de la souris }
   procedure wmlbuttondown(var msg: tmessage); virtual wm_first + wm_LButtonDown;
   { wmrbuttondown sera exécutée si on appuie sur le bouton droit de la souris }
   procedure wmrbuttondown(var msg: tmessage); virtual wm_first + wm_RButtonDown;
end;
procedure T_application.initmainwindow;                 {Méthode de T_application }
begin
  mainwindow:= new(p_fenetre, init(nil, 'Petit programme'));
end;
procedure T_fenetre.wmlbuttondown(var msg: tmessage);   { Méthodes de T_fenêtre }
begin
  messagebox(hwindow, 'Bouton gauche souris', 'Message', mb_ok);
end;
procedure T_fenetre.wmrbuttondown(var msg: tmessage);
begin
  messagebox(hwindow, 'Bouton droit souris', 'Message', mb_ok);
end;
var mon_appli: T_application;                           { Variable globale }
begin                                                   { Programme principal }
  mon_appli.init('Wmini');
  mon_appli.run;
  mon_appli.done;
end.
```

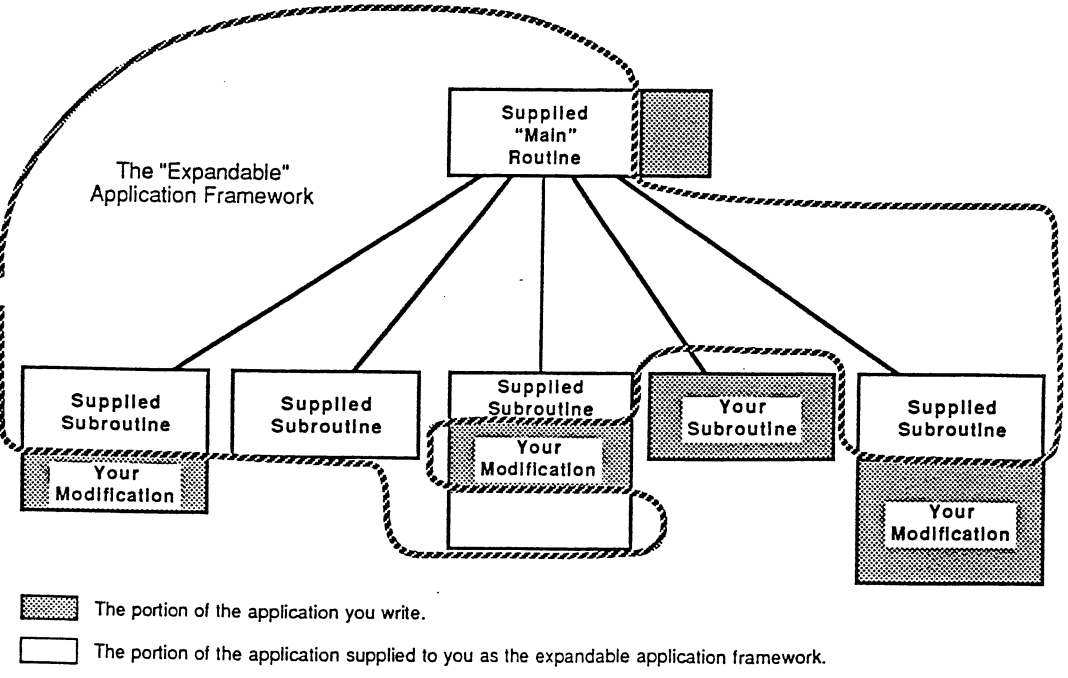
```
var mon_appli: T_application;          { Variable globale }
begin                                  { programme principal }
  mon_appli.init('Wmini');
  mon_appli.run;
  mon_appli.done;
end.
```

Messages	Valeurs	Constantes
Messages de fenêtre	$0000	wm_first
Messages utilisateur	$0400	wm_user
Messages d'identificateur	$8000	if_first
Messages d'usage interne	$8f00	id_internal
Messages de notification	$9000	nf_first
Messages d'usage interne	$9f00	nf_internal
Messages de commande	$A000	cm_first
Messages d'usage interne	$FF00	cm_internal

```
Function SendMessage(Wnd: HWnd; Msg, wParam: Word; lParam: Longint): Longint;
```







MacApp Class	Basic Responsibilites	Relationship to the other MacApp classes
The Class TApplication	• Manage all interactions between the application and the Finder • Manage the event queue • Display the menu bar • Handle all the menu commands	• Creates TDocument objects • Informs document objects when to perform certain actions
The Class TDocument	• Manages everything to do with documents (files) that are owned by the application, including: - opening documents - closing documents - saving documents to disk - reverting to the previous version of a document	• Creates TView objects • Creates TWindow objects • Creates TFrame objects • Informs view objects and window objects when to perform certain actions
The Class TWindow	• Manages everything to do with windows including: - opening windows - closing windows - moving windows - resizing windows - activating windows - deactivating windows	• Informs the frame objects that comprise the window when to perform certain actions like scrolling
The Class TFrame	• Manages everything to do with portions of windows including: - scrolling - resizing - coordinate transformations (Note that frame objects correspond closely to QuickDraw grafPorts in that each has its own coordinate system, pen state, shading pattern, font, etc.)	• Informs the view object that it displays when to perform certain actions
The Class TView	• Manages everything to do with rendering the images seen in a window or frame, especially drawing the image. • Highlights the current selection • Tracks the mouse • Prints the image • Assists in pagination of the image	• Creates TCommand objects
The Class TCommand	• Manages everything to do with user interactions that affect the data in the document, including - mouse interactions and feedback - mouse commands - menu commands - keyboard input • Implements the undo and redo functions • Interacts with the clipboard, as required	• Informs a TDocument object when to perform certain action

Figure 4-2 The roles of the basic MacApp classes in the MacApp framework.

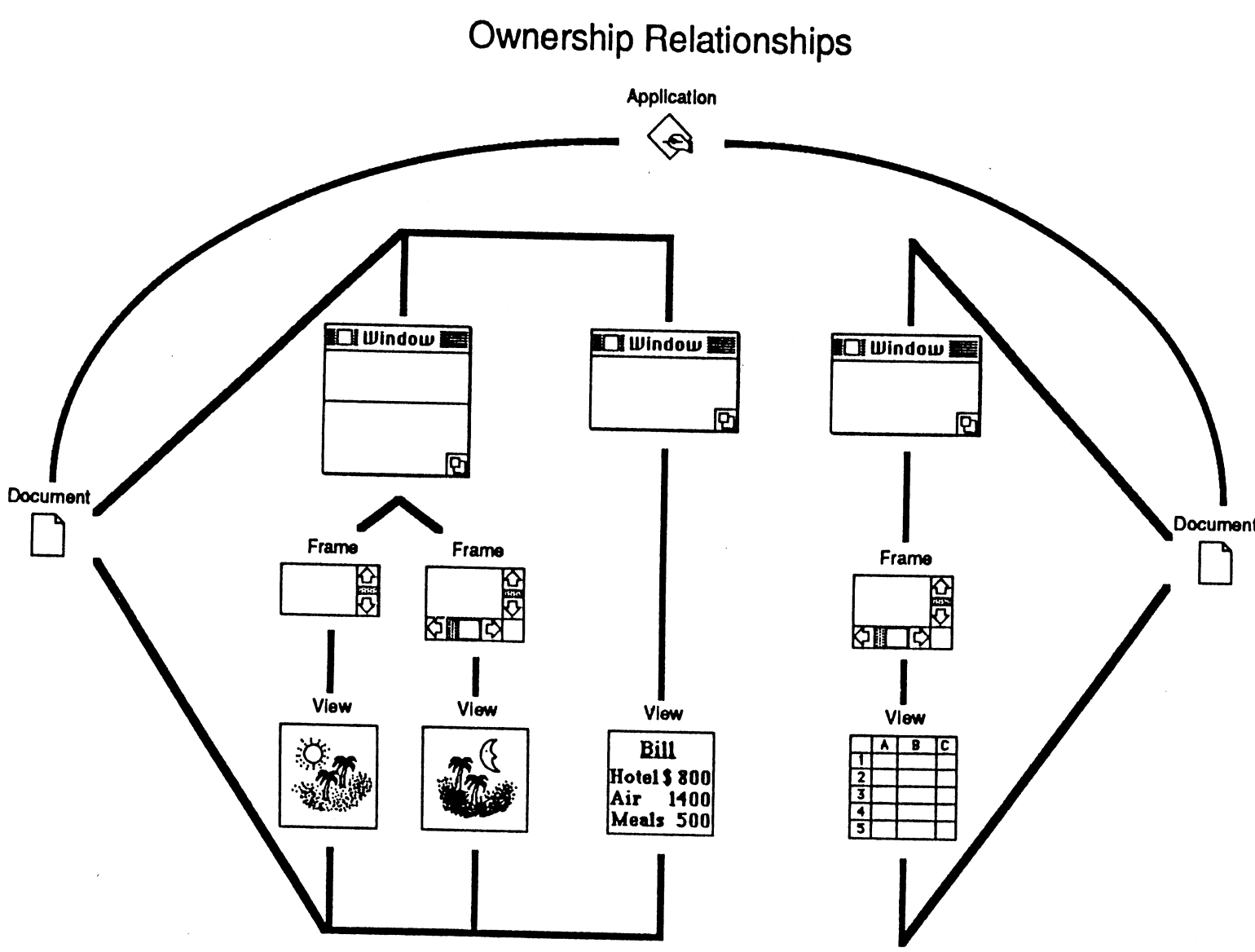


Figure 4-3 The ownership hierarchy between the various objects of the MacApp framework.